Class Objective/Overview

• Familiarize with *Stack Applications – Function Call*
• Understand with *Stack Applications – Expression Evaluation*
• Understand with *postfix Expression and the conversion from infix*
• Understand *Queues*
• Understand *Bounded Queue*
• Understand *Priority Queue*
Stack Applications:
Function Calls
Using Stacks in Function Calls

• We call the collection of information that represents a function call in progress an activation of the function.

• Computer systems use a runtime stack (called the activation stack) to keep track of the return addresses, actual parameters, and local variables associated with function calls.

• Each function call results in pushing an activation record containing that information onto the stack.

• Returning from function is accomplished by getting and saving the return address out of the top record on the stack, popping the stack once, and jumping to the saved address.
Using Stacks in Function Calls

• Activation stack is very helpful in executing recursive functions

• Computers really don’t do recursion. Instead, a recursive algorithm really gets translated as a *series of stack pushes* followed by a *jump back* to the beginning of the recursive function.

• Using a stack makes it easy to “backtrack” to the proper location when a function call returns.

• **Recursive functions** require large activation stack memory size. Systems with limited activation stack size (e.g., Windows 3.1 system) recursive functions may overflow the system.

• A similar problem can occur when programming for *embedded systems*. Such systems generally have very limited memory overall, including very limited activation stacks.
Using Stacks in Function Calls

Example:
Calculation of 3!

```c
int fact(int n)
{
    if (n<=1) return 1;
    else
        return n * fact(n-1);
}
void main( )
{
    int value;
    value = fact(3);
}
```

Stack contents during each recursive call

ARI = activation record instance
Using Stacks in Function Calls

Example (cont’d):
Calculation of 3!

```c
int fact(int n)
{
    if (n<=1) return 1;
    else return n * fact(n-1);
}

void main()
{
    int value;
    value = fact(3);
}
```

Stack contents during each `return`
Stack Applications: Expression Evaluation
Using Stacks in Expression Evaluation

- **Postfix** (also known as Reverse Polish Notation or RPN) is a parentheses-free notation for mathematical expressions:

  - Operators in postfix appear after their operands. Examples of postfix expression versus the conventional *infix* notation:
    - $1 \ 2 \ +$ instead of $1+2$
    - $1 \ 2 \ 3 \ * \ +$ instead of $1+2*3$
    - $1 \ 2 \ + \ 3 \ *$ instead of $(1+2)*3$

- **Infix** notation
  - A binary operator appears between its operands (e.g., “$1+2$”).
  - More complex than postfix, because it requires the use of operator precedence and parentheses (e.g., “$(1+2)*3$”).
  - In addition, some operators are left-associative, and a few are right-associative (e.g., “$-3$”).
Using Stacks in Expression Evaluation

• Postfix is easy to evaluate using a single stack to hold operands:

  • When you see a constant, push it onto the stack
  • When you see an operator that needs k operands,
    • pop k numbers from the stack
    • apply the operator to them
    • push the result back onto the stack
  • At the end a single value remains on the stack. This is the value of the expression.
Using Stacks in Expression Evaluation

**Postfix:** no parentheses, no precedence

<table>
<thead>
<tr>
<th>Infix</th>
<th>Postfix</th>
</tr>
</thead>
<tbody>
<tr>
<td>2+3*4</td>
<td>234*+</td>
</tr>
<tr>
<td>a*b+5</td>
<td>ab*5+</td>
</tr>
<tr>
<td>(1+2)*7</td>
<td>12+7*</td>
</tr>
<tr>
<td>a*b/c</td>
<td>ab*c/</td>
</tr>
<tr>
<td>(a/(b-c+d))*(e-a)*c</td>
<td>abc-d+/ea-<em>c</em></td>
</tr>
<tr>
<td>a/b-c+d<em>e-a</em>c</td>
<td>ab/c-de<em>ac</em>-</td>
</tr>
</tbody>
</table>
Using Stacks in Expression Evaluation

Postfix Evaluation Example: \[ \text{expression } 2 \ 3 \ + \]

Scan of Expression and Action  

1. Identify 2 as an operand.  
   Push integer 2 on the stack.

   \[ \text{operandStack} \begin{array}{c} \text{2} \end{array} \]

2. Identify 3 as an operand.  
   Push integer 3 on the stack.

   \[ \text{operandStack} \begin{array}{c} \text{3} \\ \text{2} \end{array} \]

3. Identify + as an operator  
   Begin the process of evaluating +.

   \[ \text{operandStack} \begin{array}{c} \text{3} \\ \text{2} \end{array} \]

4. getOperands() pops stack twice and assigns 3 to right and 2 to left.

   \[ \text{operandStack empty} \]

5. compute() evaluates left + right and returns the value 5. Return value is pushed on the stack.

   \[ \text{operandStack} \begin{array}{c} \text{5} \end{array} \]
Using Stacks in Expression Evaluation

Evaluate the postfix expression: \(636 + 5*9/-\)

Evaluate the following postfix expressions:

i) \(AB + C*D +\)  (if \(A=2, B=3, C=4, D=5\))

ii) \(ABC^/DE^*+AC^*-\)  (\(A=27, B=3, C=2, D=3, E=17\))
Algorithm 3.6 Evaluation of a Postfix Expression

evaluationofpostfix(s, postfix)

1. Set i = 0, RES=0.0
2. While (i < number_of_characters_in_postfix)
   If postfix[i] is a whitespace or comma
      Set i = i + 1 and continue
   If postfix[i] is an operand, push it onto the stack
   If postfix[i] is an operator, follow these steps:
      i. Pop the top element from stack and store it in operand2
      ii. Pop the next top element from stack and store it in operand1
      iii. Evaluate operand2 op operand1 and store the result in RES (op is the current operator)
      iv. Push RES back to stack
   End If
   Set i = i + 1
End While
3. Pop the top element and store it in RES
4. Return RES
5. End
Infix to Postfix Conversion

(1) Fully parenthesized expression

\[ \frac{a}{b} - c + d * e - a * c \rightarrow (((a / b) - c) + (d * e)) - (a * c) \]

(2) All operators replace their corresponding right parentheses.

\[ (((a / b) - c) + (d * e)) - (a * c) \]

(3) Delete all parentheses.

\[ ab/c-de*+ac*- \]
Infix to Postfix Conversion

INFIX

\[ x - y \ast z \]

POSTFIX

\[ x \]
Infix to Postfix Conversion

\[
x - y * z
\]

The operands of ‘-’ are NOT yet in postfix, so ‘-’ must be temporarily saved in stack.
Infix to Postfix Conversion

INFIX

\[ x - y \times z \]

POSTFIX

\[ xy \]
Infix to Postfix Conversion

**INFIX**

\[ x - y * z \]

**POSTFIX**

\[ xy \]

The operands of ‘*’ are NOT yet in postfix, so ‘*’ must be temporarily saved in stack and

*Should be restore before ‘-’*
Infix to Postfix Conversion

INFIX

\[ x - y * z \]

POSTFIX

\[ xyz \]
Infix to Postfix Conversion

**INFIX**

```plaintext
x − y * z
```

**POSTFIX**

```plaintext
xyz*−
```
Suppose, instead, we started with

\[ x*y-z \]

After moving ‘x’ to postfix, ‘*’ is temporarily saved, and then ‘y’ is appended to postfix. What happens when ‘-’ is accessed?

<table>
<thead>
<tr>
<th>INFIX</th>
<th>POSTFIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x * y - z )</td>
<td>( xy )</td>
</tr>
</tbody>
</table>
The ‘*’ must be moved to postfix now since both operands of ‘*’ are in postfix.

Then, the ‘-’ must be saved temporarily. After ‘z’ is moved to postfix, ‘-’ is moved to postfix and we are done.

**INFIX**  
\[ x \times y \, - \, z \]  

**POSTFIX**  
\[ xy\ast z- \]
Infix to Postfix Conversion

• Analysis:
  
  • *Operands* are in same order in infix and postfix
  
  • *Operators* occur later in postfix

• Strategy:
  
  • Send operands straight to output
  
  • Send higher precedence operators first
  
  • If same precedence, send in left to right order
  
  • Hold pending operators on a stack
Infix to Postfix Conversion

• Conversion to postfix is generally performed via a stack. For example, you can use this pseudo code to convert a "normal" infix algebraic expression without parentheses into postfix.

• Try this with an expression like "1 + 2 * 3 + 4".
  - The output will be: "1 2 3 * + 4 +"

• This algorithm can be modified, without too much trouble, to work with parentheses as well.

//Expression with no parentheses
S = an empty stack ;
while (more input available ) {
  read the next token , T;
  if T is a variable name or a number
    print it ;
  else { // T is an operator
    while (S is not empty and T has lower precedence than top operator on S) {
      print top operator on S ;
      pop S ;
    }
    push T onto S
  }
}
while (S is not empty) {
  print top operator on S ;
  pop S ;
}
Infix to Postfix Conversion

INFIX

\[ a / b - c + d * e - a * c \]

POSTFIX

\[ a \]

Operator Stack
Infix to Postfix Conversion

**INFIX**

\[
a / b - c + d * e - a * c
\]

**POSTFIX**

\[
a
\]

/  
Operator Stack
Infix to Postfix Conversion

INFIX

a / b - c + d * e - a * c

POSTFIX

ab

Operator Stack
Infix to Postfix Conversion

INFIX

\[ a / b - c + d * e - a * c \]

POSTFIX

\[ ab/ \]

Operator Stack
Infix to Postfix Conversion

INFIX

\[ \frac{a}{b} - c + d * e - a * c \]

POSTFIX

\[ ab/c \]

- Operator Stack
Infix to Postfix Conversion

**INFIX**

\[ a \div b - c + d \times e - a \times c \]

**POSTFIX**

\[ ab/c- + \]

Operator Stack
Infix to Postfix Conversion

INFIX

a / b - c + d * e - a * c

POSTFIX

ab/c-d

Operator Stack
Infix to Postfix Conversion

INFIX

a / b - c + d * e - a * c

POSTFIX

ab/c-d

*  
+  
Operator Stack
Infix to Postfix Conversion

**INFIX**

\[ a / b - c + d * e - a * c \]

**POSTFIX**

\[ ab/c-de \]

* Operator Stack
Infix to Postfix Conversion

**INFIX**

a / b - c + d * e - a * c

**POSTFIX**

ab/c-de*+-

Operator Stack
Infix to Postfix Conversion

INFIX

\[ a / b - c + d * e - a * c \]

POSTFIX

\[ ab/c-de*+a \]
Infix to Postfix Conversion

INFIX

a / b - c + d * e - a * c

POSTFIX

ab/c-de*+a

Operator Stack
Infix to Postfix Conversion

INFIX                  POSTFIX

\[ a / b - c + d * e - a * c \]  \[ ab/c-de*+ac \]

*  
-  
Operator Stack
Infix to Postfix Conversion

INFIX

\[ a / b - c + d * e - a * c \]

POSTFIX

\[ ab/c-de*+ac*- \]
Infix to Postfix Conversion

What about Parentheses?

- Left parenthesis $\rightarrow$ push, but with lowest precedence
- Right parenthesis $\rightarrow$ keep popping and appending to postfix until left parenthesis ‘(’ is popped, drop ‘(‘ and proceed.

Example

<table>
<thead>
<tr>
<th>INFIX</th>
<th>POSTFIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x \times (y + z)$</td>
<td>$xyz+*$</td>
</tr>
</tbody>
</table>

Operator Stack
Infix to Postfix Conversion

**INFIX**

\[ x \ast (y + z - (a / b + c) \ast d) / e \]

**POSTFIX**

x

Operator Stack

\[ x \]
Infix to Postfix Conversion

INFIX

\[ x \times (y + z - (a / b + c) \times d) / e \]

POSTFIX

\[ xyz \]

Operator Stack

+ 

( 

* 

Operator Stack
Infix to Postfix Conversion

**INFIX**

\[ x \times (y + z - (a / b + c) \times d) / e \]

**POSTFIX**

\[ xyz+\]

\[-\]

\[(\]

\[*\]

Operator Stack
Infix to Postfix Conversion

**INFIX**

\[ x \times (y + z - (a / b + c) \times d) / e \]

**POSTFIX**

\[ xyz+ab \]

Operator Stack
Infix to Postfix Conversion

**INFIX**

\[ x \ast (y + z - (a \div b + c) \ast d) \div e \]

**POSTFIX**

\[ xyz+ab/ \]

Operator Stack

+  
(  
(  
-  
*  
)  
)  
)
Infix to Postfix Conversion

**INFIX**

\[ x \times (y + z - (a / b + c) \times d) / e \]

**POSTFIX**

\[ xyz+ab/c \]

**Operator Stack**

\[ + \]
\[ ( \]
\[ ( \]
\[ - \]
\[ \]
Infix to Postfix Conversion

**INFIX**

\[ x \times (y + z - (a / b + c) \times d) / e \]

**POSTFIX**

\[ xyz+ab/c+ \]

Operator Stack
Infix to Postfix Conversion

INFIX

\[ x \times (y + z - (a / b + c) \times d) / e \]

POSTFIX

\[ xyz+ab/c+d \]
Infix to Postfix Conversion

INFIX

\[ x \times (y + z - (a / b + c) \times d) / e \]

POSTFIX

\[ xyz+ab/c+d*- \]

*  

Operator Stack
Infix to Postfix Conversion

**INFIX**

\[ x \ast (y + z - (a \div b + c) \ast d) \div e \]

**POSTFIX**

\[ xyz+ab/c+d*-* \]

/  
Operator Stack
Infix to Postfix Conversion

**INFIX**

\[ x \times (y + z - (a / b + c) \times d) / e \]

**POSTFIX**

\[ xyz+ab/c+d*-*e/ \]

Operator Stack
Stack Implementations

• Array based
  • Where is top?
  • How are elements added, removed?

• Linked List based
  • Where is top?
  • How are elements added, removed?

• Efficiency of operations
  • The amortized time of a push operation is $O(1)$
Comparison of stack Implementations

• The code for the vector version is very similar to the implementation in the C++ standard library.

• By using delegation, we avoid having to implement the operations ourselves and are assured of O(1) performance.

• Using the standard containers has a space penalty.
  • vector allocates additional space to provide amortized constant insertion time.
  • list uses a double linked list which has unneeded pointers.

• Using a single linked list allocates space for the links, but these are necessary.

• Using a single linked list also provides O(1) performance.
Queues
Queues

- Like vectors and lists, queue is also an ordered collection, but the construction and access rules are purposely limited.

- A Queue is a FIFO (First in First Out) data structure. Elements are inserted in the Rear of the queue and are removed from the Front.

- Only the front (least recently inserted) and back (most recently inserted) element may be accessed at any given time.
  - Actually, many authors would limit access to only the front element.

- In general, queues are used when things must be processed "in order", but can “pile up” before we get to them.
miniQueue Implementation

miniQueue<int> miniQ;  // declare an empty queue

Queue Statement                      Queue
miniQ.push(10)                       10
       front back

miniQ.push(25)                       10 25
       front back

miniQ.push(50)                       10 25 50
       front back

n = miniQ.front()  // n = 10

miniQ.pop()                        25 50
       front back

List Statement                        List
qlist.push_back(10)                  10
       front back

qlist.push_back(25)                  10 25
       front back

qlist.push_back(50)                  10 25 50
       front back

return qlist.front()                 // return 10

qlist.pop_front()                    25 50
The `bqueue` implements the queue ADT by using an array with two indices that reference the front and the back of the queue.

The array has a fixed number of elements specified by the constant `MAXQSIZE`.

Think of the queue as a circular sequence where elements enter in a clockwise fashion.

The exit point for an element is the slot designed by `qfront`, and the entry point for a new element is at the slot identified by `qback`.

```
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Insert elements A, B, C

```
<table>
<thead>
<tr>
<th>C</th>
<th>B</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Remove element A

```
<table>
<thead>
<tr>
<th>C</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Remove element B

```
<table>
<thead>
<tr>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>
```

Insert element D,

```
<table>
<thead>
<tr>
<th>D</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Bounded Queue (bqueue)

- Treating the array as a circular sequence involves updating qfront and qback to cycle back to the front of the array as soon as they move past the end of the array (MAXQSIZE)
- Move qback forward: \( qback = (qback + 1) \mod \text{MAXQSIZE} \)
- Move qfront forward: \( qfront = (qfront + 1) \mod \text{MAXQSIZE} \)

**Array View**

```
qback  qfront
```

Insert element D, Insert element E

**Circular View**

```
D
C
qback
```

Insert element D, Insert element E

```
D
C
qfront
```

Insert element D, Insert element E
Priority Queue

• Problem: Given a collection of elements that carry a numeric "score", find and remove the element with the smallest [largest] score. New elements may be added at any time.

• This collection is called a priority queue
  • Similar to queues, it is used to simulate objects awaiting a service.
  • But instead of FIFO, the processing order is determined by the object’s individual "priority" or score.
std Priority Queue Interface

• Check the size of the priority queue or ask if it is empty.
• Look at the top (largest) element.
• Remove the largest element by popping.
• Push a new element into the priority queue. Unlike pushing onto a stack or queue, however, the element does not automatically become the first or last thing we will next retrieve, but depends on its priority value.
Priority Queues via Ordered Insert

• Implement priority queues using sorted sequential structure data structures (array, vector, or list).

• The priority queue push and pop operations would be $O(n)$ and $O(1)$, respectively.

• To push a new element onto the priority queue, we must do an ordered insert, which we know is $O(n)$.

• To pop, we remove its smallest element, which we have stored at the end of the vector with the pop_back operation in $O(1)$ time.

• If we had used descending order, the largest element would be at the front instead of at the back, and removing it would be $O(n)$.

• Note that, if we used lists instead of vectors, we could store the items in either ascending or descending order and get an $O(1)$ pop, but push would remain $O(n)$. 
std Priority Queue Implementation

• Priority Queue could use a data structure called “heap” as underline storage.
• A heap is a special version of a binary tree in which each parent node has a value that is $\geq$ the value of its two children (maximum heap) or $\leq$ the value of the children (minimum heap).
• The pop() operation removes the root node. A heap has an adjustHeap() operation that organizes the tree.
• Push and Pop have $O(\log_2 n)$
Application of Priority Queues

• Priority queues are often useful in scheduling. For example, we simulate real-world systems as a series of events that are scheduled to take place at some future time.

• All scheduled events could be kept in a (min) priority queue ordered on the time at which the event is scheduled. The main logic of the simulation is simply:

```java
while (simulation has not ended) {
    get next event from the priority queue;
    trigger that event;
}
```

• Each distinct type of event would have its own "trigger" function, many of which are likely to add one or more additional events to the priority queue.
The Radix Sort

- Order ten 2 digit numbers in 10 bins from smallest number to largest number. Requires 2 calls to the sort Algorithm.

- Initial Sequence: 91 6 85 15 92 35 30 22 39

- Pass 0: Distribute the cards into bins according to the 1's digit ($10^0$).
The Radix Sort

- Sequence after Pass 0: 30 91 92 22 85 15 35 6 39
- Pass 1: Distribute the cards into bins according to the 10's digit ($10^1$).

- Final Sequence: 6 15 22 30 35 39 85 91 92
The Radix Sort

// support function for radixSort()
// distribute vector elements into one of 10 queue using the digit corresponding to power
// power = 1  ==> 1's digit
// power = 10 ==> 10's digit
// power = 100 ==> 100's digit
// ...
void distribute(const vector<int>& v, queue<int> digitQueue[], int power)
{
    int i;

    // loop through the vector, inserting each element into the queue (v[i] / power) % 10
    for (i = 0; i < v.size(); i++)
        digitQueue[(v[i] / power) % 10].push(v[i]);
}

// support function for radixSort()
// gather elements from the queues and copy back to the vector
void collect(queue<int> digitQueue[], vector<int>& v)
{
    int i = 0, digit;

    // scan the vector of queues using indices 0, 1, 2, etc.
    for (digit = 0; digit < 10; digit++)
The Radix Sort

```cpp
// collect items until queue empty and copy items back to the vector
while (!digitQueue[digit].empty())
{
    v[i] = digitQueue[digit].front();
    digitQueue[digit].pop();
    i++;
}
}

void radixSort(vector<int>& v, int d)
{
    int i;
    int power = 1;
    queue<int> digitQueue[10];

    for (i=0; i < d; i++)
    {
        distribute(v, digitQueue, power);
        collect(digitQueue, v);
        power *= 10;
    }
}
```
Dequeues
What is a Dequeue?

• A **Deque** (double-ended queue, pronounced “dee-cue”) is a sequence of data such that
  • data can be added only at the front or at the end
  • data can be removed only from the front or the end
  • data can be examined only at the front or the end

• As the queue, the dequeue can grow to arbitrary size.

• The C++ standard deque (pronounced “deck”) is a generalized dequeue:
  • data can be added only at the front or at the end
  • data can be removed only from the front or the end
  • data can be examined at any arbitrary position

• The deque interface is identical to that of vector, except for
  • Two new operations: push_front(), pop_front() → O(1)
  • The operations capacity() & reserve(size_type) are not available.
Implementing Dequeues

• Most implementations of deque are based on a scheme that technically meets the O(1) requirements for the push and pop operations only in an amortized sense.

• This implementation is based upon the idea of allocating “blocks” (fixed-sized arrays) of elements and then using a “map” array (an array of pointers to these blocks) to keep track of the blocks.
Implementing Dequeues

The deque itself has several data members.

• **theSize** records the number of elements currently in the deque.
• **blockmap** is a pointer to the map array (on the heap).
• **mapSize** indicates the size of the map array.
• **firstBlock** is the index of the first occupied position in the map array.
• **firstElement** is the index of the first occupied position in the first block.
• **numElementsInBlock** indicates the size of the blocks.
Implementing Dequeues

The deque itself has several data members.

- **theSize** records the number of elements currently in the deque.
- **blockmap** is a pointer to the map array (on the heap).
- **mapSize** indicates the size of the map array.
- **firstBlock** is the index of first occupied position in map array.
- **firstElement** is the index of the first occupied position in the first block.
- **numElementsInBlock** indicates the size of the blocks.

```cpp
template <class T>
class deque
{
    public:
        ...
    private:
        size_type theSize;
        T** blockmap;
        size_type mapSize;
        size_type firstBlock;
        size_type firstElement;
        const static size_type BlockSize = 4096;
        static size_type numElementsInBlock;
        // Double the map size
        void deque<T>::expandMap();
    struct dqPosition {
        size_type blockNum;
        size_type elementNum;
    };
    // where do we find element # n?
    dqPosition indexAt (deque<T>::size_type n) const;
};
```
begin() & front()

- begin() and front() are pretty straightforward.
- To find the first element in the deque, we simply look in blockmap[firstElement]. That points us to the block containing the first element. Then within that block, we look at the firstElement position.
- begin() is similar in theory, but slightly more complicated because we need to use that same reasoning to construct an iterator.

```cpp
template <class T>
T& deque<T> :: front ()
{
    return blockmap[firstBlock][firstElement];
}

template <class T>
const T& deque<T> :: front () const
{
    return blockmap[firstBlock][firstElement];
}
```
We can split the problem of finding the $i$th element into cases:

- If $i$ is less than `numElementsInBlock - firstElement`, then the element we are looking for is in the first block.
- If it’s not in the first block, then we can find out how many map positions past `firstBlock` by dividing $i - (numElementsInBlock - firstElement)$ by `numElementsInBlock`. Use the resulting quotient to index into the map and find the desired block. Then use the remainder of that division as the index within the block.

```cpp
template <class T>
deque<T>::dqPosition
deque<T>::indexAt (deque<T>::size_type n) const
{
    dqPosition pos ;
    pos .blockNum = firstBlock ;
    if (n < numElementsInBlock - firstElement)
    {
        pos .elementNum = n + firstElement ;
    }
    else
    {
        n -= numElementsInBlock - firstElement ;
        ++pos .blockNum ;
        int k = n / numElementsInBlock ;
        pos .blockNum += k ;
        pos .elementNum = n - k * numElementsInBlock ;
    }
    return pos ;
}
```
indexing

- indexAt stores the result into a simple structure with two fields, blockNum and elementNum.
- Once we have that utility function, the implementation of the indexing operators becomes trivial.

```cpp
template <class T>
T& deque<T>::operator [] (deque<T>::size_type n)
{
    dqPosition<T> pos = indexAt (n);
    return blockmap[pos.blockNum][pos.elementNum];
}

template <class T>
const T& deque<T>::operator [] (deque<T>::size_type n) const
{
    dqPosition<T> pos = indexAt (n);
    return blockmap[pos.blockNum][pos.elementNum];
}
```
**back() & end()**

- The `indexAt` utility makes for a simple implementation of the `back()` function and would also help in implementing `end()`.
- Calculation involved in getting to the $i^{th}$ element in a deque is $O(1)$.
- Deques tend to have more overhead, but it’s in the constant-multipliers. In a big-O sense, deques are always no slower than arrays and vectors, and may be faster for some operations (e.g., inserting at the front of the deque).
- There are practical benefits to having at least one sequential container that does not keep all of its data in one huge contiguous area.
  - Some operating systems place rather stringent limits on the largest amount of contiguous space you can allocate on the heap.

```cpp
template <class T> 
T& deque<T>::back ( )
{
    dqPosition pos = indexAt ( theSize - 1);
    return blockmap[pos.blockNum][pos.elementNum];
}
template <class T> 
const T& deque<T>::back ( ) const
{
    dqPosition pos = indexAt ( theSize - 1);
    return blockmap[pos.blockNum][pos.elementNum];
}
```
Inserting Data into a deque

• Pushing a new element onto either the front or the back would be no problem for the deque if there is a room in both the first and the last block for additional elements.

• If the last/first block is full?
  • need to allocate space for a new block,
  • putting the pointer to the new block in the map just after/before the currently occupied slots,
  • and placing the new element at the beginning/end of the newly created block.
Inserting Data into a deque

• What if the map is full, and there’s no room in the map for any more pointers?
  • we need to create a new, larger map:
• The fill and copy operations are \( O(\text{mapSize}) \), and mapSize could be as large as theSize/numElementsInBlock, so this means that a push_back or push_front on a deque with \( N \) elements is really \( O(N) \).
• This worst case cost does amortize to \( O(1) \) if we start with an empty deque and push \( N \) successive elements into it.
• expandMap only copies pointers, not actual elements. The number of elements copied by a push call remains \( O(1) \).

template <class T>
void deque<T>::expandMap()
{
    T** newMap = new T*[2*mapSize];
    fill_n(newMap, 2*mapSize, (T*)0);
    copy(blockmap+firstBlock, blockmap +mapSize, newMap+mapSize/2);
    delete [] blockmap;
    blockmap = newMap;
    firstBlock = mapSize/2;
    mapSize *= 2;
}
Questions?
Assignment #4

• Due Tue Oct 15th, 11:59pm

• Written Assignment
  • Ford & Topp, Chapter #7, & #8:

<table>
<thead>
<tr>
<th>Question #</th>
<th>Page #</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.21</td>
<td>381</td>
</tr>
<tr>
<td>8.11</td>
<td>427</td>
</tr>
<tr>
<td>8.15</td>
<td>428</td>
</tr>
<tr>
<td>8.17</td>
<td>429</td>
</tr>
</tbody>
</table>

• Programming Assignment
  • Ford & Topp, Chapter #7 & #8:

<table>
<thead>
<tr>
<th>Question #</th>
<th>Page #</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.32</td>
<td>382</td>
</tr>
<tr>
<td>8.28</td>
<td>432</td>
</tr>
</tbody>
</table>
Assignment #4

• Due Tue Oct 15th, 11:59pm
• Submission Format:
  • Written Assignment
    • Create single PDF file with name: cs361_assignment_4_<firstName>_lastName
    • Have a cover page with your name and your email
    • Submit through Blackboard.
  • Programming Assignment
    • Make sure your two programs compile and execute using g++ on Dept’s Linux machines.
    • Create a “Readme.txt” file for each program that list how to compile and execute the program. Include your name and your email.
    • Your main file should be named “main.cpp”.
    • You should have a Makefile.
    • Zip all files (.h, .c, Makefile, etc.) for each program and name them assign4_program1 & assign4_program2 respectively.
    • Submit through Blackboard.
  • Final Submission Materials (3 Files):
    • One PDF for written assignment.
    • Two ZIP file for the two programs